High-Performance FPGA-Based QR Decomposition

Huy Nguyen, James Haupt, Michael Eskowitz, Birol Bekirov, Jonathan Scalera,
Thomas Anderson, Michael Vai, Kenneth Teitelbaum
MIT Lincoln Laboratory
Email: hnguyen@ll.mit.edu

Introduction

This paper discusses two high-performance implementations of QR decomposition suitable for demanding adaptive beamforming radar applications, where both processing throughput and latency are important. In a beamforming application, the radar typically collects a small set of samples in training mode to build an internal mathematical model of the environment. The adaptive weights are then computed from the model using the QR decomposition process in a split-second time frame. Then, in radar processing mode, the weights are applied to real-time data streaming through at very high rate.

The QR decomposition and weight computation described in this paper are implemented in the same FPGA that performs beamforming operation rather than in an external microprocessor as in conventional approaches. This integration eliminates off-chip communication in accessing training samples and updating beamforming weights.

The first QR decomposition implementation features a software-oriented approach, where a 32-bit microprocessor soft-core (Xilinx MicroBlaze) is built from random FPGA logic gates to run a C program that implements the McWhirther systolic array. The program is kept short to fit entirely inside the FPGA limited on-chip memory so that no external memory is needed.

The second design features a hardware massively parallel approach, utilizing a novel architecture that combines the McWhirter algorithm [Song] with a linear folded systolic structure [Walke] that operates with 100% processor efficiency. The folded linear structure enables the FPGA to implement large arrays in time-slice fashion with up to 50% less resources compared to the triangle array structure. To boost throughput, computation of 1/sqrt() in the McWhirter array is implemented with a non-recursive table-based approach so that no iterative computation is required, which allows the array to be fully pipelined and potentially increases throughput several times over a previous recursive-based implementation [Song].

QR Decomposition

QR decomposition is a mathematical process to decompose a matrix X into a product of two component matrices Q and R such that X = Q * R, where Q is orthonormal (Q^H *Q = identity matrix) and R is upper triangular (all elements below the diagonal are zeros). The upper triangular provides a very efficient way of inverting R by back substitution rather than full Gaussian elimination.

In many applications, a covariance matrix, C, is formed as C = X^H *X to estimate statistical correlation relationship among several random variables whose snapshots are contained in the measurement matrix, X. We are interested in computing the inverse of the covariance matrix, C^{-1}, which is not trivial for a large matrix C. However, by expressing X in term of Q and R, it is possible to reduce the computation complexity significantly.

$$C = X^H * X$$
$$= (Q * R)^H * (Q * R) = R^H * Q^H * Q * R$$
$$= R^H * R$$

Thus, C^{-1} can be found by

$$C^{-1} = R^{-1H} * R^{-1},$$

where R^{-1} can be computed efficiently from R, which is obtained from the QR decomposition of measurements X.

McWhirter Array

There are many ways of performing QR decomposition. Software-based implementations often favor Householder algorithm, which works well for centralized memory storage and accessing. Parallel processing implementations, however, require the data to be distributed throughout different processing units to support parallel execution. McWhirter array is a distributed processing algorithm that only requires near-neighbor data communication. Figure 1 shows the signal flow-graph of the McWhirter algorithm.

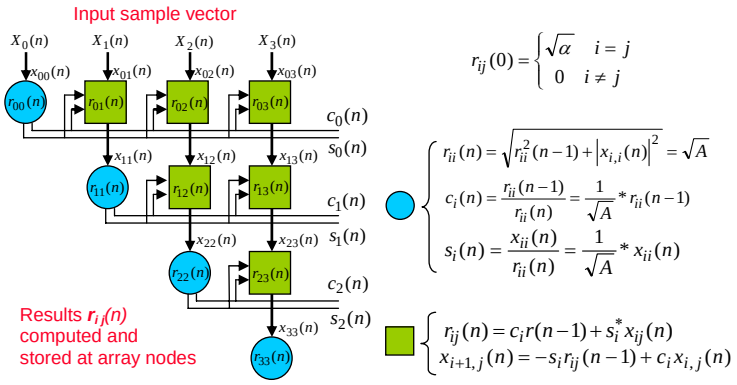

Figure 1: McWhirter Systolic Array Algorithm

Computation of 1/sqrt(A)

Most of the computations involved in the McWhirter array are rather simple, except for sqrt(A) and 1/sqrt(A). Note that since the quantity sqrt(A) can be obtained by a simple multiplication A * 1/sqrt(A), we only need to do one difficult computation, i.e., 1/sqrt(A).

In [Song], Newton's iterative approximation was used to compute 1/sqrt(A). It typically takes 3-5 iteration loops to converge to a 19-bit precision value. While simple to implement, this approach does not allow the processing node to be pipelined because of the iteration loop.

To enable pipelining, we introduced a new piece-wise linear approximation approach. The entire input space is divided into many segments, each is associated with a set of

*This work is sponsored by Defense Advanced Research Projects Agency, under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the Department of Defense.

two parameters, namely slope and offset, for a linear approximation of 1/sqrt(A) within the segment. The number of segments required is based on the desired level of the error. The two parameters are stored in two tables indexed based on A. With some optimization, 1/sqrt(A) uses only a fraction of the logic and memory in the FPGA.

Software-Oriented Implementation

We set out to explore the maturity of the FPGA embedded processor technology. As a pilot project, we implemented the McWhirter array in C and ported that to a Microblaze soft-core made up from random logic in the FPGA.

There were several tool and development board related issues, but within one month and a half, we were able to get the QR decomposition running on the Microblaze, with clock rate was in excess of 100 MHz. The program used floating-point variables to obtain high precision, one advantage of using C. Although a floating-point library was available, we coded up our own 1/sqrt() function based on a fast routine used in the game Quake3 by Carmack. This resulted in a speed up of 4 over the standard C call to sqrt().

Beside the full software implementation, we also explored different software-hardware trade-off levels, such as the FPGA logic gates to implement the "internal nodes" which perform only simple computation (square node in Figure 1), leaving the "boundary nodes" with 1/sqrt(A) to the processor. Since there are many internal nodes, the parallelization boosts up the throughput significantly.

Hardware-Array Implementation

The McWhirter algorithm has a triangle signal flow-graph, but unfortunately it is often not possible to implement the full triangle in a FPGA because of resource limitation. The straight-forward choice is to implement only the first row, and re-use it to perform the computation in the other rows as we step through the array schedule. Unfortunately, this approach results in many nodes idling when used on the lower rows of the array, bringing overall efficiency to about 50%. A mapping proposed by [Walke] folds the array such that two datagrams are overlapped at one time, allowing the idle processors to operate on the 2nd datagram while the first is till in process. This results in 100% efficiency.

Firgure 2 shows the McWhirter array with node indexing that indicates the order of processing. The mapping requires that the number of internal nodes be even, so a dummy internal column has been added. The two extra columns on the right are for computing adaptive weight vectors using the same array in back-substitution.

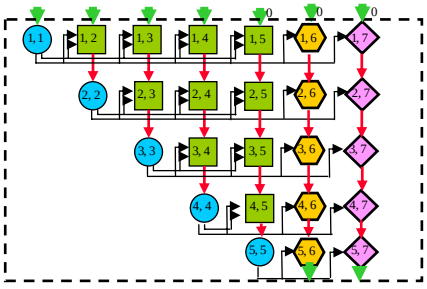

Figure 2: Triangle Array Schedule

Figure 3 shows the folded array, where the execution of the green datagram is overlapped with the brown from the previous time frame, and orange from the next time frame.

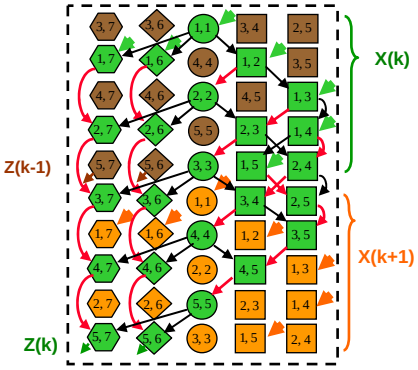

Figure 3: Folded Linear Array Schedule

Figure 4 shows a set of processors to implement one time-slice of the folded array schedule. The multiplexers are used to change the red data path from one schedule step to the next. Note that we only use Walke's folded array mapping, but not his computation nodes.

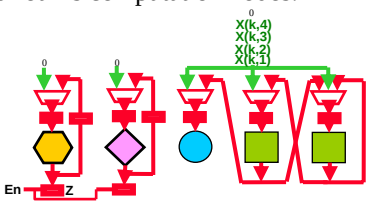

Figure 4: Linear Array Processors

Conclusion

We have presented two implementation of the QR decomposition process. One implementation is software based, using the 32-bit microprocessor made from random logic in the FPGA. This approach is simple, since most mathematical functions come with a standard C compiler. The 2nd implementation is a massively parallel hardware array with 100% processor utilization. This implementation is suitable for deep-pipelining to the clock level (as opposed to just schedule level as in [Song]).

With the slowing of Moore's law, performance improvement cannot be bet on clock rate doubling every 1.5 years. Rather, more novel algorithms and architectures will be needed to improve throughput. An optimized system is costly in terms of development and debugging. We see the use of the embedded microprocessor in the FPGA as a great help for developing complex FPGA-based systems.

References

[Eskowitz] Eskowitz, M., Gleyzer, V., Haupt, J., Mirhosseini, R., "Adaptive Beamforming Using FPGA Embedded Microprocessors," Major Qualifying Project Report, Worcester Polytechnic Institute, 2004.

[Song] Song, W.S, Rabinkin, D.V., Vai, M.M., Nguyen, H.T., "VLSI Bit-Level Systolic Sample Matrix Inversion," MIT Lincoln Laboratory Report NTP-2, 2001.

[Walke] Walke, R., "Adaptive Beamforming Using QR in FPGA," High Performance Embedded Computing Workshop, 2002.